Programming Errors

There are four categories of programming errors:

1. Compilation (syntax) errors:
   - error in the form of the statement: misspelled word, unmatched parenthesis, comma out of place, etc.
   - detected by the compiler (at compile time)
   - computer cannot correct error, so object program is not generated and thus program is not executed.
   - computer (compiler) prints error messages, but continues to compile

2. Linker errors

   - linker errors prevent the generation of an executable image
   - common linker errors:

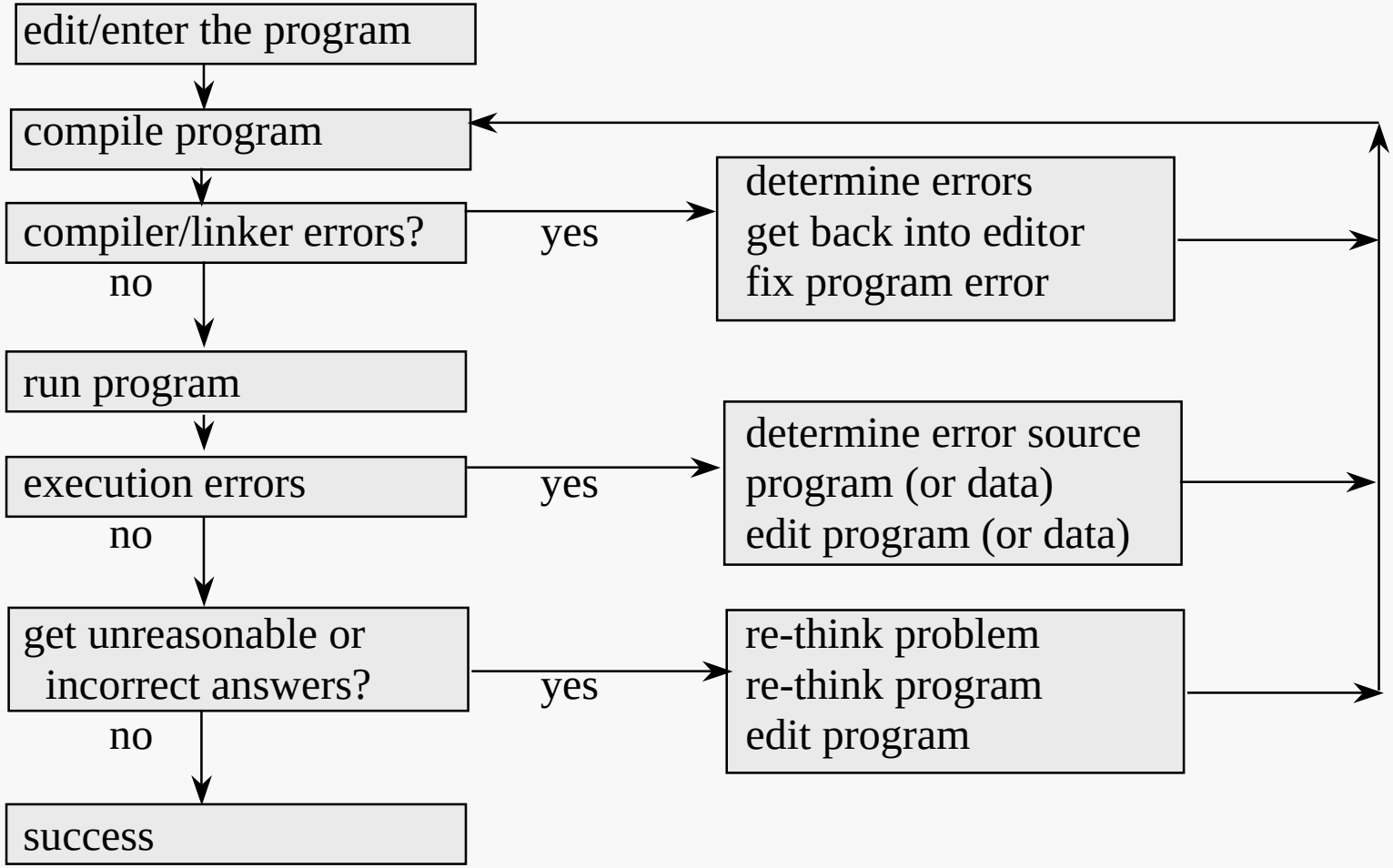
     * specifying the wrong header file
     * disagreement among the function prototype, function definition and calls to that function

Programming Errors (cont)

3. Execution (runtime) errors
   - program compiles OK, but something goes wrong during execution... like division by 0, or an attempt to read data that does not exist, etc.
   - detected by the computer run time system
   - computer (usually) prints error message and stops !!

4. Logic errors
   - program compiles and executes OK, but produces unexpected or incorrect results
   - detected by programmer (you !!)
   - hardest to find

Program Development Process



Background

C++ is based on C; in fact, C is almost a subset of C++

Developed by Bjarne Stroustrup at AT&T Bell Laboratories

Standard C++

Was formally adopted by American National Standards Institute and ISO to provide a common base in 1998 - our course is based on C++ as represented by “The C++ Programming Language”, Third Edition by B. Stroustrup

Unambiguous and machine independent.

C++ implementations for a specific machine or operating system usually provide "extensions" to the standard.

The use of language extensions is controversial. Extensions provide added functionality but decrease the portability of source code.

Skeleton Program

- Directives
- Header
- Declarations
- Comment
- Body of Program

```
#include <iostream.h>
#include <stdlib.h>

int main( )
{
int  total;    // comment
char letter;


//  This program is complete




cout << "Hello world!!" << endl;
return EXIT_SUCCESS;
}
```

Syntax and Semantics

■ syntax - (grammar) rules that specify how valid instructions (constructs) are written

■ semantics - rules that specify the meanings of instructions.

The syntax of an assignment statement requires that you have:

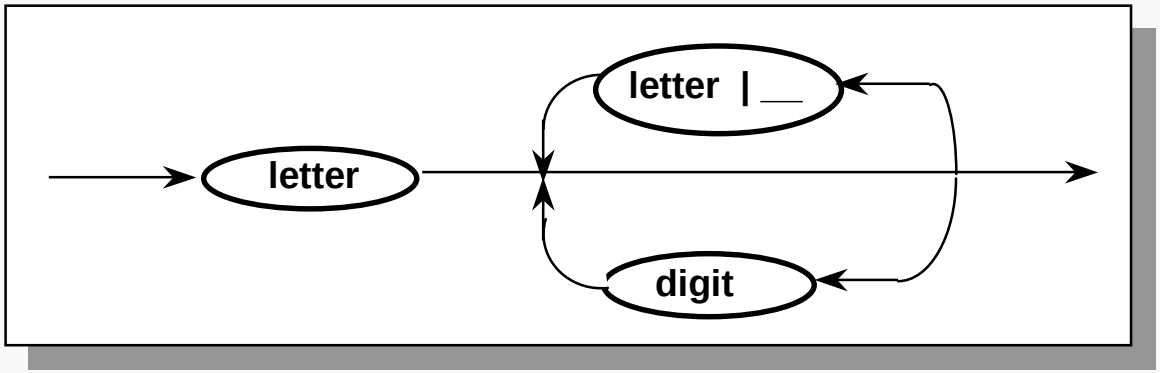
variable = expression;

The semantics of an assignment statement determines that the value of the expression on the right is stored in the variable on the left. For example:

sum = sum + score;

■ syntax diagram:

–identifier



C++ keywords are strictly lower-case; choose identifier names that are mixed.

Identifiers

Identifiers are names of objects.

Identifiers can be used to name locations in the computer's memory where

1) changeable values (variables) can be stored and
2) where unchangeable values (constants) can be stored.

Valid and invalid C++ identifiers:

```
40hrsperweek      three.onefour      to day       N/4
3.14              name$              X-Ray        R2D2
SumOfGrades       x2                 night        C3pO
two_sevenone      double              2_71        INT
```

Identifiers should be more than one or two letters and descriptive!

Directives

Compiler directives do not generate machine code

- directions to the compiler about how to carry out the compile step

Examples:

```
#include <iostream.h>
```

- include text from a file named `iostream.h`

```
#define NUMBER 100
```

- replace `NUMBER` by `100` everywhere `NUMBER` occurs

The Simple Data Types

C++ has only four simple data types (classes of data): integer, real, character and logical

- integer - positive or negative whole number
  - e.g.: +5280, -12, 100
  - three subtypes: `short`, `int` and `long`
- real - decimal number
  - e.g.: 98.6, 3.14, -3.4561E-12 , 3. , .4 // permitted, but 3.0 and 0.4 are better
  - three subtypes: `float`, `double` and `long double`
- character - alphanumeric (alphabetic & numeric) and special symbols
  - e.g.: 'A', 'q', '%', '8', '\n', '\0'
  - one subtype: `char`
- Boolean - a logical value (`true` or `false`)
  - one subtype: `bool`

Constants

C++ has two types of constants:

Literal constants - explicit numbers or characters

```
16, 45.5f, "Freddy", 'M', 3.14159
```

Named constants - location in memory, identified and referenced by a name, where a data value that cannot be changed is stored.

Constants are named in the constant definition (declaration) section of a C++ program:

Syntax:

const type identifier = literalvalue ;

Declarations/definitions are non-executable statements.

EX:

```
const int   EMPTY   = 0;
const int   FULL    = 1;
const float PI      = 3.14159f;
const char  NL      = '\n';
```

By default real constants are of type double. Immediately succeeding a real constant with an f or F, (case insensitive), causes the constant to be of type float.

† It is common programming practice to CAPITIALIZE constants.

Standard Constants

Program Exit Status Constants

```
#include <stdlib.h>      //location of exit status constants

EXIT_SUCCESS
EXIT_FAILURE
```

Use to indicate the return status of main()

Implementation Limits

```
#include <limits.h>      //location of implementation constants

INT_MIN   INT_MAX        // integer bounds
LONG_MIN  LONG_MAX       // long bounds


#include <float.h>       //location of float constants

FLT_MIN   FLT_MAX        // float bounds
DBL_MIN   DBL_MAX        // double bounds
```

Statements

The line

```
cout << "Hello world!!" << endl;
```

of (3.5) is an example of a statement. This is another name for an instruction to the computer.

All statements must end with _____.

The semicolon in C++ is a _____.

Variables

■ variable - a location in memory, referenced by name, where a data value that can be changed is stored.

– variable declaration section

Syntax:

```
type  identifier;
type identifier1, identifier2 ;
type identifier = InitialValue ;
```

Note the lack of formatting. C++ is a free-format language. Formatting conventions are described in the document: "*ELEMENTS OF PROGRAMMING STYLE*"

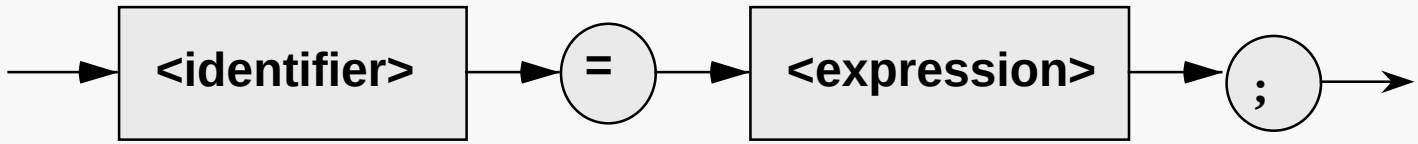
Ex:

```
int    Weight,
       Height,
       Length = 0 ;
float  ClassAverage,
       qca ;
char   MiddleInitial ;
double PercentCorrect = 0.0 ;
```

C++ requires that all identifiers be declared before they are used.

Assignment

Syntax



Ex: `total = ( total + 1 ) / count;`

An assignment statement is an executable statement. It gives a variable the value of an expression.

The memory location referenced by variable name total gets the value of the (simple) expression. The value of the expression on the right is stored in the variable on the left.

Statements such as:

```
X = X + 1 ;
```

in Fortran, C++ or BASIC (where '=' is the assignment operator) make perfect sense in programming, albeit not in mathematics (where '=' denotes equality).

Assignment (cont)

- The value of an assignment statement is _____.
- The type of the expression should conform to _____.

float var = integer value;	int var = float value;
legal - integer will be	inadvisable - warning by most compilers
'promoted' to a float	float value will be truncated

Consider the previous declaration (3.13). Which of these are valid?

```
Weight        =  Height;   // doesn't make logical sense
Qca           =     3.5;
MiddleInitial =     'L';   // MiddleInitial equals character L
PercentCorrect =    .78;
Weight        =     Qca;   // stores truncated value, 3
MiddleInitial =       a;
ClassAverage  =  Weight;   // integer value is "widened" to
                           //    a float, 3.0
```

Increment/Decrement

VarName = VarName + 1; occurs so often that C++ gives a special unary operator, the increment operator ++. It can be used as

VarName++ or ++VarName

The difference is very subtle: assume VarName = 1 and consider

Assignment	Number	VarName
Number = ++VarName;	2	2
Number = VarName++;	1	2

The semantics is that the prefix version evaluates the right hand side after incrementing and the postfix version evaluates before incrementing.

Similarly, there is a special unary decrement operator: **--**.

It can also be used as

VarName-- or --VarName

with similar semantics.

Warning – mixing of ++ , -- operators in aruthmetic expressions should be avoided.

Increment/Decrement (cont)

Given * denotes multiplication, then if:

```
i = 3 ;          k =  5 ;
j = 8 ;          m = -2 ;
```

evaluate:

```
m = ++i * k--;
m = j++ * k--;
m = j++ * ++k;
m = --i * --k;
m = --j * k++;
m = i-- * j--;
m = ++i * ++j;
m = i-- * ++k;
m = --i * j++ * ++k;
```

m	i	j	k

Note: Expresssions of these forms can be avoided. For example,

```
m = ++i * k--;
```

Can be expressed more clearly as:

```
++i;
m = i * k;
k--;
```

Arithmetic Expressions

Combine constants and variables with valid operators to form expressions.

The type of the expression is based on the operations performed and the operand types.

Decimal values and integers can be combined using all arithmetic operators. Integer values are converted to decimal and resulting expression type is float or double, as appropriate:

```
float  X = 9.0 / 12;   // value 0.75 is stored
double Y = 5 / 2.0;    // value 2.5 is stored
double Z = 5 / 2;      // value 2.0 is stored (Why?)
```

Beware: 7 / 12 is evaluated using integer division and so is equal to 0.

Types of arithmetic expressions:

1. A single numeric constant or variable
2. Arithmetic expression preceded by + or - (unary sign)
3. Two arithmetic expressions "infixed" by arithmetic operator
4. Arithmetic expression enclosed in parentheses

Operator Hierarchy

The order in which operations are performed within an expression follows strict precedence rules:

First, expressions in parentheses are evaluated.
Then: (1) - (negation)
(2) * / % (remainder)
(3) + -

Operators in the same group are evaluated left to right.

Examples:

```
21 - 3 * 2 - 47 + 5          21 / 3 * 4
     Result: ____            Result: _____
```

Parentheses needed (that is, do they affect the answer)?

```
( X + Y ) * Z                X + ( Y * Z )
```

Parentheses are often added for reading clarity even when not required.

Assignment and Expressions

Recall that the same variable can appear on both sides of the assignment operator:

```
x =  x + y ;
```

Two common uses of variables:

1. Counting (determining number of input data)
2. Summation (running subtotal)

Counting example:

```
numstudents  =  numstudents + 1;
```

Summation example:

```
total = total + score ;
```

Note on Summation:

Could be used to compute:

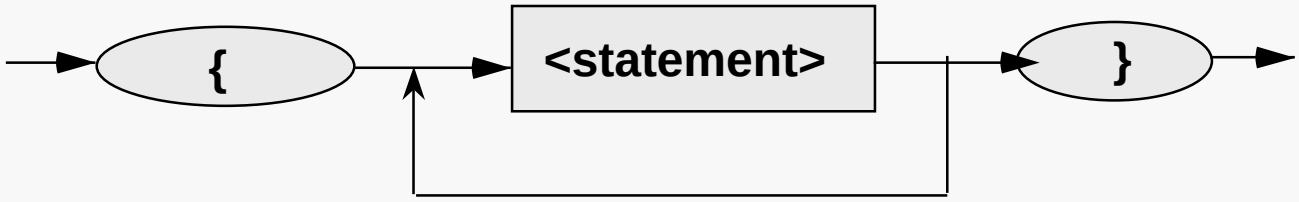
$$\sum_{i=1}^{N} i$$

However,

$$\sum_{i=1}^{N} i = \frac{N (N + 1)}{2}$$

Compound Statement

A collection of statements surrounded by { } delimiters.




```
{
    cout << "Enter a number: " << endl ;
    cin >> num ;
    temp = num ;
    cout << "Thank you very much." ;
} // no semicolon is required at the end of
  // a compound statement
```

Where statement is one of:

null statement (;)

assignment statement

cout or cin statement

compound statement

A compound statement can be used wherever a statement is called for in a syntax diagram.

Standard Functions

Can be thought of as additional operations on specific data types.

function_name (parameter_list)

where each parameter can be an arithmetic expression.

Examples:

Determine the difference between two integers, a and b

```
diff = abs ( a - b );  // math.h
```

Get the length of a string

```
len = strlen("There is always time to do it over");
                                        // string.h
```

Get the square root of a number

```
rootx = sqrt( x );    // math.h
```

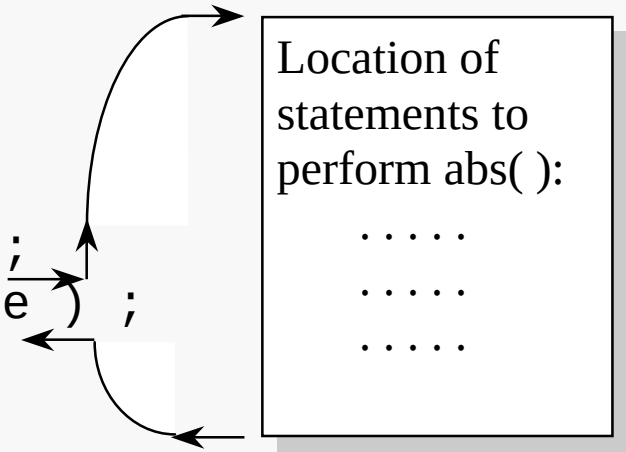
The use of standard library functions requires the inclusion of specific standard header files.

Function Invocation

Function (ex. abs) *'calls / invocations'* temporarily interrupt the default sequential statement-to-statement *flow of control*.

Example:

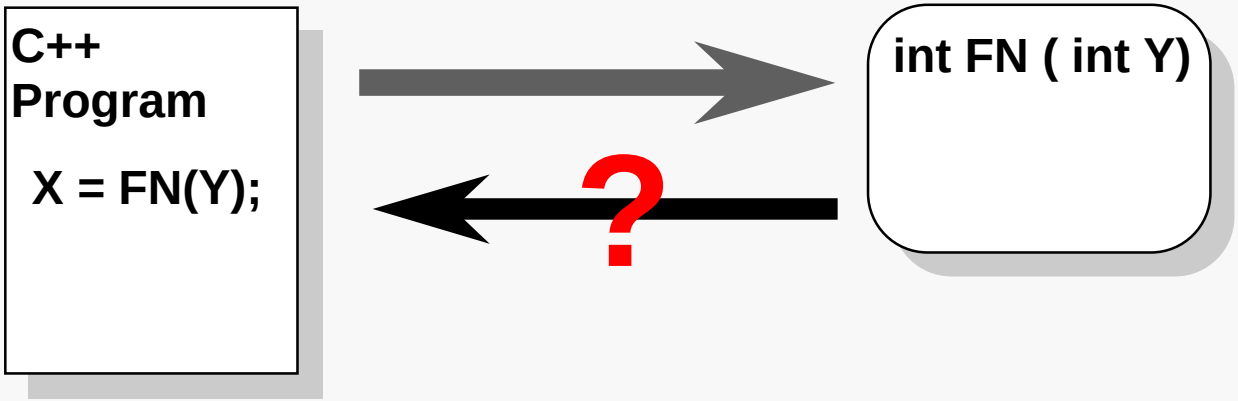
```
——————
currAge   = nowYear - birthYear + 1 ;
ageDiff   = abs ( currAge - clientAge ) ;
cout  << ageDiff  << endl;
——————
```



Normal flow of control resumes after the called routine completes execution (returns).

Function Communication

Parameter Lists - mechanism by which a C++ program 'passes' data to functions.



Communication with standard functions is two-way, or duplex, where the data passed back by the function occurs through the return value or through reference parameters.

Example:

```
ageDiff   = abs ( yourAge - myAge ) ;
      // = abs ( myAge - yourAge ) ;
```